UNIVERSITY of York

This is a repository copy of *XRound : A reversible template language and its application in model-based security analysis*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/54730/

Version: Submitted Version

Article:
Chivers, Howard Robert orcid.org/0000-0001-7057-9650 and Paige, Richard F. orcid.org/0000-0002-1978-9852 (2009) XRound : A reversible template language and its application in model-based security analysis. Information and Software Technology. pp. 876-893.

https://doi.org/10.1016/j.infsof.2008.05.006

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

XRound: A Reversible Template Language and its application in Model-Based Security Analysis

Howard Chivers and Richard F. Paige
Department of Information Systems, Cranfield University, Shrivenham, UK.
Department of Computer Science, University of York, UK.

hrchivers@iee.org, paige@cs.york.ac.uk

Abstract Successful analysis of the models used in Model-Driven Development requires the ability to synthesise the results of analysis and automatically integrate these results with the models themselves. This paper presents a reversible template language called XRound which supports round-trip transformations between models and the logic used to encode system properties. A template processor that supports the language is described, and the use of the template language is illustrated by its application in an analysis workbench, designed to support analysis of security properties of UML and MOF-based models. As a result of using reversible templates, it is possible to seamlessly and automatically integrate the results of a security analysis with a model.

Keywords Model-Driven Development, Template Processing, UML, Security Analysis

1. Introduction

Transformations are a critical component of Model-Driven Development, particularly in the Model-Driven Architecture (MDA) [1]. To this end, the Query/Views/Transformations (QVT) [2] standard has been developed, in order to provide a precise mechanism for describing transformations between models.

QVT transformations can be *unidirectional* (i.e., from one metamodel to a second not necessarily different metamodel) or *bidirectional* (i.e., reversible between two metamodels). The former is of critical importance in MDA, e.g., for transforming platform independent models (PIMs) into platform specific models (PSMs). The latter is vital for supporting round-trip engineering, and also *rigorous analysis* of models: the results of a logical analysis (e.g., correctness, timing analysis, security analysis) may need to be reflected in the source of a transformation. For example, a static analysis may be applied to a PSM, resulting in changes being made to that PSM. These changes may need to be reflected in the original PIM.

Limited tool support currently exists for bidirectional transformations; key state of the art is summarised in Section 2. Bidirectional transformations can be awkwardly implemented by the sequential application of unidirectional transformations, but this is not entirely satisfactory because information – e.g., detailed representations of platforms, model element identities – may be lost after each unidirectional transformation is applied. This will particularly be the case with transformations that are reused from libraries in unexpected ways. More generally, it is difficult to ensure round-trip consistency in a sequence of unidirectional transformations.

This paper describes a new *template*-based language, called *XRound*, for specifying bidirectional transformations between models of arbitrary languages. Moreover, the paper presents powerful tool support for this language that implements bidirectional transformations, particularly to support *merging* of the results of model analysis with an original model. XRound and its tool support are both illustrated in the context of a case study demonstrating a particular form of model analysis, namely *security risk analysis* (explained in Section 1.1). General lessons learned about model analysis and using XRound in the context of merging analysis results with models are also extracted and discussed.

1.1 Context and contribution

Given a model of a system, such as a Unified Modelling Language (UML) or Matlab/Simulink/Stateflow model, we can apply tools to analyse the model, to determine if it has desirable properties. A variety of analyses are possible and are used in practice, particularly *consistency checking* (e.g., does the information contained in one UML diagram contradict that contained in a second diagram), *timing* (e.g., is the worst-case execution time for a system satisfactory), *failures* (e.g., does the system mitigate for a particular class of fault) and *security*. We illustrate the use of XRound and bidirectional transformations in the context of security risk analysis in this paper.

Security risk analysis is concerned with discovering threat paths in a system which allow potential attackers to access system assets. Concrete security objectives are in the form of specific unwanted outcomes to particular assets

(e.g. integrity of particular data); however, the analysis process is forced to consider all possible threat paths to each asset, and this requires an efficient analytic tool.

The basic idea behind any form of model analysis is as follows. A model (e.g., in UML) is annotated with properties. In the case of security risk analysis, the properties attached to a model are predicates: facts that are true, or that we wish to assert, about the model (examples follow in the sequel). These properties are collected by an analytic tool, which then processes them and returns results. The results of analysis may be additional properties that need to be integrated with the original model, e.g., new security requirements that have been added to control threats, failure modes for collections of components, worst-case execution time for a subsystem. This integration is generally quite difficult, especially because engineers may need to use different tools for modelling (e.g., a UML tool) and specialised analyses.

The usual solution for lightweight tool integration of this form is *template processing*. A template processing system applies a template to a data model via a template processor, resulting in the extraction and formatting of the data for some particular application. An example of a template language is XSLT, which is used for transforming XML documents, usually into text or HTML.

This is an attractive solution, since it allows designers to use their preferred modelling environment, and does not necessarily require a complete definition of the languages supported by that environment (e.g., a complete UML metamodel). It is also preferable from the tool software perspective; for example, type checking of properties can be implemented once within the analytic tool, rather than in each UML environment. It is also attractive from the perspective of compatibility with the traditional principles and practices of MDA and MDD. MDA, for example, operates in terms of application of model transformations to elaborate models, add platform details, remove details, and eventually generate code. Template processing is another mechanism for model transformation; in particular, when applied to security analysis (as we do in this paper), template processing supports the concept of *in-place transformation*, which updates a model to include new (security) information. Thus, a template processing approach to model analysis adds no additional complexity to typical MDA/MDD processes, and in addition helps to support domain experts (e.g., security engineers) in their efforts.

Template processing provides an important bridge between different tools, but the currently available solutions are unable to support the reverse path of unifying the output data back into its original source. Round-trip engineering of analysis results back into the UML is therefore not straightforward with a conventional template processor, but is an important requirement for specialist analytic tools,

XRound is designed to overcome this problem. Its objective is to maintain the advantages of template processing, including simple scripting of data transformations and independence between input and output applications, while supporting bidirectional transformations. This language and its supporting template processor allow model analysis tools to import Extensible Markup Language (XML) models with a source-specific metamodel, and re-generate the XML when the analysis model is changed. The security analysis workbench application described in Section 7 is one such application: a specialised analysis tool that imports UML models in the XML Metadata Interchange (XMI) format, and uses a bidirectional transformation to merge its results into the original XMI model.

The contribution of this paper is to describe the XRound language, its motivation, its relationship with standard template processing, and how it supports bidirectional transformations via so-called *reversible templates*. Additionally, the implementation and use of the XRound language is shown to be feasible by presenting a supporting template processor and a practical application. XMLSource, a Java-based template processor for XRound, is described in both system and implementation terms, and the successful use of XRound in security analysis is discussed. General lessons learned about supporting different kinds of model analysis are also synthesised in the conclusions.

We commence with an overview of related work on transformations and model merging, and then in Section 3 discuss the notion of a reversible template, which is at the foundation of XRound. The required processing for reversible templates is presented. Section 4 presents XRound itself, and Section 5 illustrates the language with several small examples. Section 6 summarises the template processor that supports XRound, and section 7 explains the use of XRound in supporting security analysis. We then discuss lessons learned, conclusions, and future work.

2. Related Work

There is substantial related work on model transformation, model merging, and template-based techniques that is relevant to the approach presented in this paper. We now review this work.

2.1 Transformations

Transformations are a critical component of Model-Driven Development, particularly in the MDA [1]. The MOF (Meta-Object Facility) 2.0 QVT standard [2] has been developed in order to provide a precise, flexible mechanism for modelling transformations. QVT provides the means for declaratively capturing both unidirectional and bidirectional model transformations. These can be independent (i.e., the result model is not linked with the source model after transformation) or dependent. Dependent transformations aim at supporting a similar approach to the reversible templates applied in this paper; we are unaware of any QVT tool support for dependent transformations at this stage. QVT aims to support a

variety of scenarios for transformation. Regeneration and reconciliation of transformation results is most similar to what is intended for the reversible templates we present.

Tools for supporting transformations have been developed. Of note amongst these are the Atlas Transformation Language (ATL) [3], XMF-Mosaic [4], Yet-Another Transformation-Language (YATL) [5], VIATRA2 [6] and Epsilon [7]. These languages and tools are all targeted at Model Driven Development. There are also transformation tools outside of the Object Management Group (OMG) standards; for example, the TXL [8] framework has some similarities to QVT, though it has been predominantly targeted at programming language transformation. In this sense, TXL has some similarities to the model-to-text proposals, such as MOFScript [9].

The generative programming community has made use of template-based techniques to implement transformations [10], and tools have emerged, including Velocity [11] or Java Emitter Templates (JET) [12]. These are generally unidirectional transformations aimed at minimizing the amount of code that needs to be rewritten in a code generation process.

Tratt's Converge meta-programming language [13] has also been used successfully to implement a transformation language, in this case as a domain-specific language. Tratt also describes a change propagating transformation, also implemented using Converge, wherein updates made to the source model are automatically propagated to the target model. A similar approach to change propagating transformations is considered by Alanen and Porres in their Coral system [14]. They describe their tool architecture in detail, focusing on its activation mechanisms, which have some similarity to the architecture of the template engine described in this paper.

Hu et al describe a programmable editor for developing structured documents (typically XML) based on bidirectional transformations [15]. Their intent is to allow operations to be applied to a document view, and to have an editor automatically derive a consistent document source along with a transformation to produce the view. In this manner, consistency is guaranteed by construction. They define a new (declarative) language for describing transformation rules, and implement a view updating scheme (similar to those from the database community) which reflects view modifications on the underlying repository.

2.2 Model merging

A model management operation related to transformation is model merging (sometimes referred to as model composition, weaving, or unification). Merging models is the process of integrating two or more models – often representing parts of the same system – into a unified, consistent, single model. Model merging is related to database schema merging. A generic approach to model merging was described by Pottinger [16]. Approaches to model merging targeted at Model-Driven Development have begun to appear. The Atlas Model Weaver (AMW) [17], which makes use of ATL, is one of the first generic prototypes. It makes use of a weaving model to describe correspondences between model elements (e.g., which elements are to be merged). The Epsilon Merging Language (EML) is a rule-based language which allows models to be compared (to identify correspondences), and elements to be thereafter merged [7]. Pierce et al's [18] research on data synchronisation is strongly related to model merging and composition, as it focuses on the more general problem of synchronising XML documents via bi-directional transformations. It is targeted specifically at efforts to ensure view consistency for tree-structured data, but does not propose a concrete language for transformations and updates, nor has it been applied directly to security analysis. By contrast, the work in the DEGAS project explored security analysis in the context of the Choreographer platform [19], but they did not present a concrete transformation language as well.

Model merging could be used to solve the problem of combining the results of an analysis of models with the models themselves. For example, a set of EML rules could be written that identify where the results of analysis should be inserted into a source model, and a second set of rules written that describe the results of the merging process. The main difference between the approach offered by model merging, and the approach in this paper, is that any such merging rules will focus on integrating the results of analysis with source models. The approach we present in this paper allows one set of templates to be written which support both source-to-target generation (i.e., producing a model to be analyzed) and target-to-source generation. Conceptually, using a model merging approach and using reversible templates are equivalent.

2.3 Security analysis

There has been some related work on model-based security analysis, beyond what we have mentioned in Sections 2.1 and 2.2.

Related to model merging and model transformation is aspect-oriented modeling and weaving. Models of cross-cutting concerns (called aspects) are woven with system models via an automated process. The application of aspect-oriented modeling to security analysis is considered by Petriu et al [20] and Houmb et al [21]. These approaches differ from the work presented in this paper by focusing more on the development of security artifacts and supporting analysis, rather than being able to reflect the results of analysis in system models.

Breu et al consider security analysis in the context of MDA [22]. They model security requirements using UML diagrams and focus on using transformations to generate security artefacts for web services systems. Jurjens [23] has presented an approach to building security-critical systems using UML, and provides tool support for automated verification. Brændeland [24] has explored security analysis for component-based systems. The focus of the work in this paper is more on the infrastructure

needed to support different kinds of security analysis without being restricted to specific tools and metamodels, on supporting the analysis of security risk in the system design process, rather than reasoning about security functionality, and on being able to support reflection of the results of analysis automatically in models.

3. Processing Overview

This section introduces the key concepts behind a reversible template, and then describes how the required processing functions motivate the overall structure of the template language.

3.1 The system context

The original motivation for reversible template processing was to enable the analysis and round-trip updating of XMI representations of UML models produced using proprietary design tools. However, the template processor and the XRound language are general XML transformation tools, a typical application of which is shown in Fig 1.

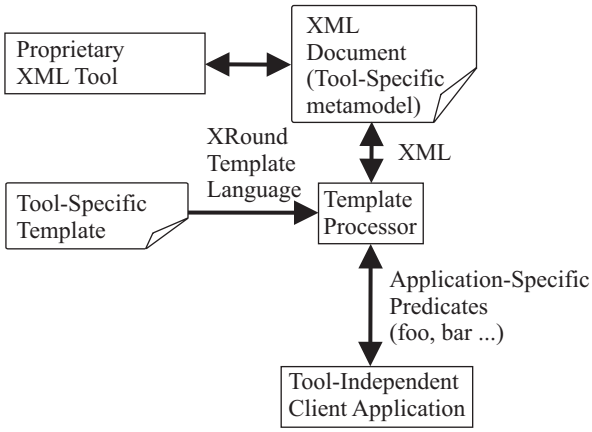

Fig. 1 Reversible template processing architecture

In Fig 1, an application is required to process and modify an XML model, which is normally managed by a proprietary tool. The application could directly manage the reference XML document, but this would dedicate the application to a proprietary metamodel. The purpose of the reversible transformation is to decouple the application from the tool-specific metamodel, by specifying a template which allows the application to import elements of the XML model, and also update that model to be consistent with changes made by the application.

One benefit of using a template, as opposed to automating the transformation between two meta-models, is that only partial metamodels need to be specified; in other words, only the elements of the proprietary model required for the application need to be understood. The benefit of using a single template to specify a bidirectional transformation is consistency: only a single document is needed to specify the relationship between the application and the proprietary metamodel.

The relationship between the template processor and the application is an application programming interface (API), in which the application is a client of the template processor. Facts about the model are *predicates*, which are exchanged between the template processor (see section 6, below) and the application.

Three main functions must be supported by the template processor; they are:

- *validation*: to check that the format of the XML document is compatible with the supplied XRound template;
- *import*: to read the XML document, and provide the application with the predicates specified in the template; and
- *export*: to update the XML document to be consistent with predicates held by the application.

One implementation of the template processor is described in section 6; the following sections describe the principles of reversible templates in more detail.

3.2 Bidirectional Transformations and Model Unification

Template processing is usually a one-way operation as shown in Fig. 2: the template processor locates elements in the input tree and publishes them, suitably formatted.

In the case of XML data, the input to the template processor is a tree; the output may be XML, or it may be published in another format such as text or HTML. Conventional templates are capable of expressing arbitrary computation, but their fundamental structure is still to navigate to selected nodes in the input tree, extract information, and produce suitably formatted output. The benefit of a template over a standard programming language is usually that it is tailored to the particular type of input and output required.

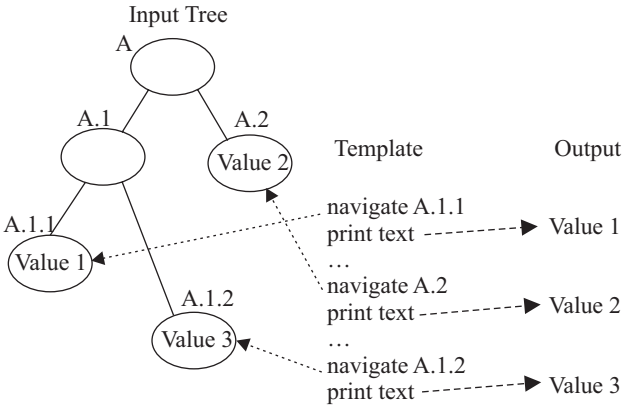

Fig. 2 Conventional template processing

Reversible templates defined in XRound are similar in structure to existing templates, but encapsulate a fundamentally different type of operation: unification. The operation of a reversible template is shown in Fig. 3.

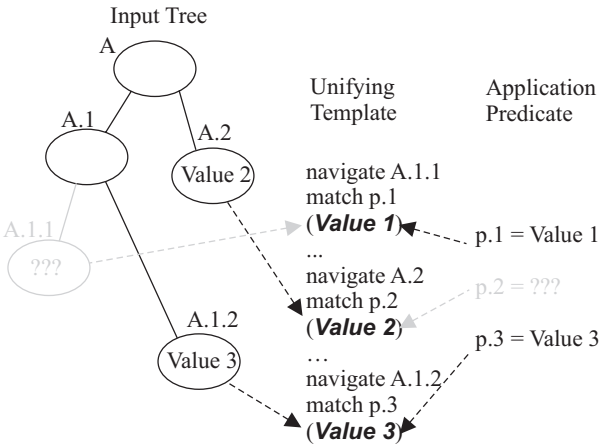

Fig. 3 The template unification process

A reversible template navigates to elements in the input tree, in a similar way to a conventional template, but it also references values that are shared with the application. The fundamental operation is to match, or *unify*, values in the source tree with values in an application predicate. Unification allows values to be determined from either the source tree, or the application predicate, or if values are set in both, to ensure that they are consistent. For example, in Fig. 3 the first value is not known in the source, but is available in a predicate; the opposite is true for the second value; and the third is the same in both source and predicate, so this unification succeeds.

The underlying unification process determines the design of the template language; each part of the template identifies a unification slot, and the fundamental operation is 'match', which is to unify the slot with either the XML input tree, or the application predicate.

Unification is conceptually straightforward, but designing a template language that exploits this process does present some problems, including:

- The source navigation for a reversible processor is not quite the same as a conventional template processor, because is has to unify input nodes that do not exist. For example, in Fig. 3 it is not simply the case that the input node does not have the first value set, but that the whole node (A.1.1) is missing. The template language must allow the programmer to specify which nodes are allowed to be missing, and which areas in the source tree are fixed. In XRound, nodes that may be missing are marked as *mutable,* and can be created by the template processor.
- Because either nodes in the input tree, or attributes within nodes, may be missing, it is not always possible to select nodes based on an attribute value, as in an Xpath [25] expression; it is necessary to unify nodes that are present while certain attribute values within the node are missing. In XRound this problem is solved by a general constraint mechanism, which constrains unification slots to specified values. Constraints are also unified as part of the matching process and can therefore be used to specify the types of predicate that can be generated, constrain XML node selection, and determine application predicates to be unified.

The underlying unification process determines some features that are needed in a reversible template language: the specification of unification slots and constraints. The next section describes template processing functions, and how they are supported.

3.3 Template Processing

This section describes the operation of template processing in sufficient detail to motivate the clause structure of the XRound template language. Section 6 describes the template processor in more detail.

The previous section described template-based unification, and this places some requirements on the top-level sections of the template language, which are known as the *clauses*. Essentially a clause must:

- specify a number of unification slots;
- allow the specification of constraining values for each slot; and
- unify values in the XML input and/or in application predicates with slot values and constraints.

In order to allow for a separate verification section, and also to allow the user to distinguish parts of the XML input that should be fixed, as opposed to those that may be rewritten, three types of clause are defined in XRound:

- *validate*
- *structure*
- *roundtrip*

A *validate* clause specifies validation checks, a *structure* clause references elements of the XML input that should not be modified, and a *roundtrip* clause includes input nodes that may be modified when the XML is regenerated from application predicates. The value of the *structure* clause is that it allows some performance optimisations compared to *roundtrip* clauses, because it does not have to account for missing nodes. However, it is not the case that all nodes visited by *roundtrip* clauses can or should be re-written; nodes that can be updated are specifically identified in XRound by a *mutable* attribute.

The three processing functions described in section 3.1 are now described:

Validation. Validation can be used to make any checks that the programmer requires, but its primary aim is to ensure that the template and the XML input are compatible. A particular template will apply to a limited range of source tools and versions; validation clauses in the template are used to make any measurements on the XML input data that are necessary to check that XML document is compatible with the template.

After the XML input and the template have been successfully opened and parsed, each *validate* clause is executed, and each must succeed for the validation to succeed. No other clauses are executed during validation, and the validation clauses are not executed as part of any other processing.

Import. The import operation is similar to normal template processing, it is used to assemble predicates from the XML input and provide them to the client application.

Any *structure* clauses are first executed, followed by *roundtrip* clauses. Each clause is unified with constraints specified within the clause, but not with any application predicates. The clauses have one or more *publish* attributes that mark completion; when these are reached the unification slots within the clause are checked and, if complete, a predicate is exported to the client.

Export. The export operation merges predicates from the client application back into the XML input, then saves the result. The purpose of the operation is to update the XML representation with any changes that have been made by the application, without the need for the application to manage the specific XML metamodel, and without the need to write different templates for input and output processing.

The first processing stage executes all the *structure* clauses in the template; although this will not result in any updates to the XML output, it is necessary because it may build reference information that is used later (see Performance Management, below). There are two further processing stages, the second removes mutable nodes, assuming that nodes no longer present in the application have been deleted intentionally, and the third re-builds nodes from the application predicates. In both cases, the operation (remove, build) takes place only for mutable nodes that have been encountered during a successful unification of a roundtrip template clause. The values written to the rebuilt nodes are obtained from the unification slots in the template, and so may contain values from the application predicates, from the XML input, and from clause constraints.

In summary, the process that allows a template to be interpreted in both directions is unification; this has implications for the types of navigation that can be carried out within a template and determines the need for other structure in each clause: unification slots and constraints. The three key operations of validation, import and export are supported by the clause structure in XRound, allowing the programmer to specify validation checks (*validate*), elements of the XML that should not change (*structure*), and parts of the XML model that may be modified (*roundtrip*).

4. The XRound Language

This section describes the complete XRound language. It begins by describing how an XRound template is organised in terms of clauses and how they support unification slots, constraints, and transformations. This is followed by a discussion of other types of constraint, and additional language features, including those that support performance management and debugging.

4.1 Basic Template Structure

The top-level structure of the template language is given in the abbreviated XML Document Type Definition (DTD) in Table 1.

Table 1 Abbreviated XRound Document Type Description

```
<!ELEMENT tpl.template ((tpl.validate|tpl.structure|tpl.roundtrip)* )>

<!ELEMENT tpl.validate ((tpl. declare|tpl.constraint)*,tpl.specification+)>
<!ATTLIST tpl.validate  length       CDATA #IMPLIED
                        auxLength    CDATA #IMPLIED>

<!ELEMENT tpl.stucture ((tpl. declare|tpl.constraint)*,tpl.specification+)>

<!ELEMENT tpl.roundtrip
          ((tpl. declare|tpl.constraint|tpl.uniqueName)*,tpl.specification+)>

<!ELEMENT tpl.declare>
<!ATTLIST tpl.declare   position     CDATA #REQUIRED>
                        name         CDATA #REQUIRED>

<!ELEMENT tpl.constraint (tpl.value+)>
<!ATTLIST tpl.constraint position    CDATA #REQUIRED>

<!ELEMENT tpl.uniqueName (tpl.value+)>
<!ATTLIST tpl.uniqueName position    CDATA #REQUIRED>

<!ELEMENT tpl.value (#PCDATA)>

…
```

An XRound template is a well-formed XML document containing three node types that may occur in any number and any order: *tpl.validate, tpl.structure* and *tpl.roundtrip.* These are the *clauses* introduced in the previous section. Attributes in each clause node specify the number of unification slots (*length* + *auxLength*), which may be indexed as an array in the subsequent template (e.g. *position = "0"*). The slots are divided into two types, the first (specified by *length*) are mapped directly to an application predicate; the second (specified by *auxLength*) are auxiliary variables used during template processing. It is necessary to index the predicate as an array, to ensure that the order is specified for the application; however, it is also possible refer to unification slots by name, for the purpose of readability. This is the purpose of the *tpl.declare* statement, which may occur within any of the three main node types. The scope of such a declaration is the node (template clause) in which it occurs, and it allows any attribute that would specify a unification slot number to use the declared name as an alternative (e.g. *position* = "UML_CLASS").

Each clause may have any number of constraints; each constraint has a *position* attribute that specifies the associated unification slot, and a number of values.

Because the round trip process can generate nodes that were not previously part of the XML input, it is sometimes necessary to generate new unique names. For example, unique identifiers (*xmi.id*) may be needed for UML elements in XMI models. *tpl.uniqueName* specifies that a unification slot will be filled with a unique name that is generated by the template processor, if it is not otherwise defined by the unification process.

A clause therefore specifies the unification space, or number of slots, and gives constrained values to those slots. One or more *specification* nodes in each clause determine the correspondence between the XML input and unification slots in the template, and hence the application predicates.

The remainder of the language is presented as constructs and examples, rather than a DTD; this is because specification nodes may quote from the DTD of the source XML document, modified with additional optional attributes; the resulting DTD for the XRound would therefore either be application specific, or unhelpfully contain extensive provision for "any" nodes.

4.2 Template Specifications

A template specification is well-formed XML; unlike some template languages it follows a tree structure, rather than a sequence. Depth in the tree indicates subsequent operations and breadth allows the specification of alternatives. There are three types of node in a template specification: *Source* Nodes, *Navigation* Nodes, and *Matching* Nodes. Certain XRound attributes may appear in more than one type of node, so these will be summarised before specification nodes are described.

4.2.1 Generic Attributes

Generic attributes may appear in several different node types; they are used to control the behaviour of the template processor.

A *publish* attribute can appear anywhere in a specification tree, and its effect is to test if unification is complete, and if so mark that result as successful. Table 2 illustrates the use of this attribute.

Table 2 The *publish* attribute

```
<first>
   <second publish="TRUE"/>
   <third>
      <fourth publish="TRUE"/>
</third></first>
```

The specification in Table 2 would find all instances of first…second and first…third…fourth that unified. (first, etc, are not of course valid node names.)

A *tpl.mutable* attribute specifies that the sub-tree beneath the node in which this attribute is set can be removed or re-written when predicates from the client application are exported back into the XML document. This attribute may only occur in *source* or *navigation* nodes, within *roundtrip* clauses; Section 5.2 provides an example that illustrates the use of *tpl.mutable.*

Two other attributes may appear anywhere within an XRound template, they are *tpl.debug* and *tpl.message.* Their primary function is for template debugging, and their use is described in section 4.5, below.

4.2.2 Source Nodes

Source nodes name a node type in the XML input document. They cause the template to evaluate all nodes of that name from the current position in the XML document. A typical template is therefore interspersed with node names from the source document, together with statements specific to the template language.

4.2.3 Navigation nodes

The XRound language supports four types of navigation statement:

- *select*: evaluate all nodes with a given name;
- *selectfromChildren*: evaluate child nodes;
- *moveUp*: move up in the XML document tree; and
- *selectRegisteredNode*: evaluate registered nodes.

Selection statements result in the evaluation of all the selected nodes. The *selectRegisteredNodes* statement is concerned with performance management, which is discussed in section 4.4, below; examples of the other three statements are given in Table 3.

Table 3 Navigation statements

```
<tpl.select node="UML:ClassifierRole">

<tpl.selectFromChildren
          node="UML:AssociationEnd"
          position="0">

<tpl.moveUp steps="2”>
```

A *tpl.select* node evaluates all nodes in the input tree with the specified node name; the example in Table 3 selects all *UML:ClassifierRole* nodes in the XML input document.

A *tpl.selectFromChildren* node selects child nodes from the present position in a specified order. Each occurrence of *tpl.selectFromChildren* specifies the position (i.e. index) and name of the child node to be selected. In this example the first occurrence of a *UML:AssociationEnd* node is selected. Note that there is no need for the template language to have a named statement that evaluates all child nodes of a given name, since that function is provided by directly quoting a source node.

The *tpl.moveUp* node moves the present position in the XML document tree up by the specified number of steps.

4.2.4 Matching Nodes

There is a single match node, *tpl.match*, within the template language; it instructs the template processor to unify an element in the XML input tree with the specified unification slot, any previously specified constraints, and depending upon the process mode, a predicate retrieved from the client application. The type of matching carried out is controlled by the *nodeType* attribute, which may take one of four values:

- TEXT_NODE
- ATTRIBUTE_NODE
- MULTIPLE_TEXT
- MULTIPLE_ATTRIBUTE

Text matching unifies with a text node, and attribute matching unifies with a selected attribute from the current node. Multiple matching parses a given text element or attribute into components, and unifies with one of the components. Table 4 gives examples of the four matching options.

Each *tpl.match* node specifies the index of the unification slot that must be matched (*position*), as either an index or a declared name. The relationship between the unification slots and the client predicates is fixed, so this does not need to be specified separately. The node to be matched from the XML input is always the current node, reached by the last navigation. The first two match types unify the value of node text data, or an attribute by name, respectively. The ‘multiple’ variants add the ability to decompose an XML value into components, and match one of those components with a unification slot.

A multiple text or attribute node type unifies one value in a list of separated values. These are used in situations where the XML text string (text node or attribute) is composite, and must be decomposed for the application. For example, given the attribute *myLunch=”fish,chips”*, the example above would correctly match the number of values in the attribute (*length*=”2 “) and attempt to unify the value ‘*chips*’ (*tagIndex=”1”*) with the template slot 3. One use of this feature is to pack and unpack UML tags with compound value elements, recorded in XMI as single text nodes.

This is the core of the reversible template language. Other language features include additional forms of constraint, and the support of performance management and debugging. These will be discussed in the following sections, after which the core language will be illustrated with worked examples.

4.3 Further types of Constraint

The primary means of constraining processing is via constrained unification. (See Sections 3.2 and 4.) In brief, *tpl.constraint* may be used to specify a list of values for a unification slot. Any valid entries for the slot must be consistent with the corresponding value in the XML document, the value in the application predicate, and also take one of the specified constraint values.

There are two other types of constraint supported by XRound:

- selection by a constant, which avoids the need to use auxiliary unification slots just to specify constants; and
- constraint matching, which allows more complex constraints than can be achieved using the primary constraint mechanism.

Table 4 Matching Options

```
<tpl.match  nodeType="TEXT_NODE" position="0">

<tpl.match  nodeType="ATTRIBUTE_NODE" attribute="name" position="1">

<tpl.match  nodeType="MULTIPLE_TEXT" tagIndex="1‟ length="2‟ position="3" >

<tpl.match  nodeType="MULTIPLE_ATTRIBUTE" attribute="myLunch‟ tagIndex="1‟ length="2‟
            position="3" >
```

Table 5 Constraint by a constant

```
<tpl.roundtrip length="1" auxLength="1">
<tpl.declare position="1" name="constrain_true">
<tpl.constraint position="1">
    <tpl.value>true</tpl.value>
</tpl.constraint>
...
<tpl.match  nodeType="TEXT_NODE" position="constrain_true">

                                  ---- alternatively ---

<tpl.match  nodeType="TEXT_NODE" text="true">
```

4.3.1 Selection by a constant

It is feasible to specify a constant using an auxiliary unification slot; however, there are cases where constraint by a constant is an aid to template readability. The syntax is to replace the position attribute in a match statement, with *text="value"*, as shown in Table 5.

Table 5 illustrates two ways of ensuring that a text node has the value 'true'. In the first an additional unification slot is declared, given a meaningful name, and constrained to the value 'true'; the slot is then referenced as required. The second uses constraint by a constant, in which the value is specified as needed. Even without the name declaration, the first is considerably longer. Of course, writing constants where they are needed is not always the best programming practice, and either can be used as appropriate. This is essentially a syntactic shortcut, which does not modify the underlying unification mechanism; however, it cannot be used where a specific unification slot is necessary; for example, to share a constant such as a predicate name with the application.

XML documents often include elements of the form *name="nnn" value="vvv*"; where the name of interest to the application is a constant, this form of constraint is particularly effective in this case.

4.3.2 Constraint Matching

The unification process described in Section 3.2 constrains unification slots independently; however, there are instances where it is necessary to accommodate dependencies between variables.

For example, consider the need to check tool names and version numbers to verify that a template is able to process the supplied XML; for example, a template may be able to accommodate XML generated from the tools and versions given in Table 6.

Table 6 Example of pair-wise constraints

```
tool = "uml_A"      version = "1.2"
tool = "uml_A"      version = "1.2a"
tool = "xmi_uml"    version = "15"
```

The core language would be able to extract the tool name and constrain it to *uml_A* or *xmi_uml,* and similarly extract the version number. However, an additional form of constraint is needed to correlate the tool to its version.

This is achieved by index matching, which specifies that two unification slots must be matched by a constraint at the same position in each constraint list. The syntax is given in Table 7:

Table 7 Index matching syntax

```
<tpl.matchConstraintIndex
            position="..."
            position2="...">
```

The two positions index unification slots, as usual, and the constraint specifies that the two slots must be filled with constraints from the same position in each constraint array. For example, the template in Table 8 implements the version checking requirement in Table 6.

The match statements in Table 8 unify the selected attributes with the possible values for tool and version, and the additional *matchConstraintIndex* test ensures that only valid pairs of values are permitted.

Although this function is limited to pairwise comparisons, it can be used to implement constraints of any order.

Table 8 Version checking template

```
<tpl.validate auxLength="2">
<tpl.constraint position="0">
    <tpl.value> uml_A </tpl.value>
    <tpl.value> uml_A </tpl.value>
    <tpl.value> xmi_uml </tpl.value>
</tpl.constraint>
<tpl.constraint position="1">
    <tpl.value> 1.2 </tpl.value>
    <tpl.value> 1.2a </tpl.value>
    <tpl.value> 15 </tpl.value>
</tpl.constraint>
<tpl.specification>
    ...
   <tpl.match nodeType="ATTRIBUTE_NODE" attribute="tool" position="0">
   <tpl.match nodeType="ATTRIBUTE_NODE" attribute="version" position="1">
   <tpl.matchConstraintIndex
         position="0" position2="1">
   ...
```

4.4 Performance Management

The main performance problem in template processing is the need to repeatedly scan all the nodes in a document. This problem can be seen in the *roundtrip* example in Table 13, in Section 5.2, below. A reference to an *xmi.id* is obtained from a node of interest, but in order to find the class name that corresponds to the reference it is necessary to scan the entire document for *UML:Class* nodes. Since Classes are in user-defined packages they can occur at any level of the XMI hierarchy, so it is not feasible to limit the search size by navigating from the tree root.

The types of node that are revisited in this way are often a relatively limited number of fixed design points; in the UML example these are primarily the classes and objects. If it were possible to simply remember the location of these nodes then these auxiliary searches could be made much more efficient. This, quite simply, is what the performance management statements in XRound implement. There are two statements, one that records fixed points, and one that navigates to previously recorded nodes. The syntax for these statements is given in Table 9.

Table 9 Performance management syntax

```
<tpl.registerNode/>
....
<tpl.selectRegisteredNode
              node="...">
```

The *registerNode* statement records the current node, and the *selectRegisteredNode* statement can then be used to search just those nodes that have been registered for a given node type.

In the example of Section 5.2, each *UML:Class* could be registered, allowing the set to be revisited later without the need to search the entire document tree. Instead of searching the document tree with a *select* statement, the *selectRegisteredNode* could be used; the result is the same, but considerably faster.

The only restriction on the use of these statements is that mutable nodes cannot be registered, and that nodes must be registered before they can be selected. Clauses in the template are executed in order, so normal practice is to register nodes in early *structure* clauses; these nodes can then be referenced in the remainder of the template.

The value of these performance features is model and template specific. However, the parsing performance of a real system model indicates the effectiveness of node registration. The system design used for test purposes is a high level model of an industrial distributed system, which was analyzed using the security analysis application described in Section 7 [26]. The size of the UML model[1], and comparative parsing times[2], are given in Table 10.

Table 10. Example of performance management benefit

Model Characteristics	
XML File Size	1.9 MB
Total Number of Classes	142
Total Number of Associations	296
Template Processing Performance	
Elapsed time with registered classes	2.1 s
Elapsed time without registered classes	36.4 s
Additional complete model searches without registered classes	751

The template used to process this model registers only the class nodes, and this simple strategy results in a substantial performance benefit (from 36.4 seconds to 2.1 seconds). A *select* operation, as opposed to a *selectRegisteredNode* operation, forces the template processor to check every node in the model for the specified attribute (e.g. class name), and without

[1] The number of classes and associations in table 1 are slightly higher than those that may be inferred from the published case study [26]; the difference is that these are for the whole model, not just the system design elements discussed in the study.

[2] Measured on a 2.81 GHz Pentium 4-based machine with 1GB of RAM.

registration this results in 751 additional full model searches. A major contribution to this cost is the processing of associations; when an association is encountered, the template references the class at each end to obtain class names from the *xmi.id* attributes specified in the association node. A similar operation is required to associate tags with classes. Not all templates would need to dereference IDs in this way, but this is a relatively common requirement, and the performance management features described here make these dereferencing operations much more economical.

4.5 Debugging

Finally, there are two important features in the language that aid template debugging: message and debug attributes, which are generic attributes that can be added to any node. Their syntax is given in Table 11.

Table 11 Generic debugging attributes

```
tpl.message=".. message text .."

tpl.debug="TRUE"
```

The message attribute can be included in any node, and sets a message for the template tree below that node. If any errors are issued during the processing of that template sub-tree, then the message will be included in the error report. It is good programming practice to include messages in every clause header; they provide useful comments and invaluable narrowing of the problem space when an error is reported.

The debug attribute is not intended to be a permanent feature of a template. Whenever a node is encountered with this attribute, the following information is printed:

- the current message (see above);
- the current predicate, which may not be fully defined;
- the template node that requested the printout;
- attributes of the template node;
- the current document node; and
- the attributes of the current document node.

This provides a compact summary of the current status of the template processor, and is sufficiently informative to trace the behaviour of a template without needing the complete processor status; in particular, it allows the unification process to be monitored. In practice, however, this level of detail is rarely required; it is often sufficient to know that a particular node in the template is reached.

When a template fails, the most common problem is detecting the node that failed to match the document, so the most common use of this debugging feature is to probe where a template succeeds or fails.

5 Examples

This section provides three examples of template clauses, which demonstrate how well the template language is able to hide round-trip processing complexity. The examples are drawn from templates that support roundtrip analysis of the security of UML system models, where the UML design tool uses XMI as its export format. The first two examples illustrate structure and roundtrip clauses; the third is a roundtrip clause for the same application, but a different source metamodel.

5.1 A structure clause

Table 12 presents a complete *structure* clause, which extracts UML Class names with given stereotypes.

There are two unification slots in the template, and these correspond directly to a client predicate with two values. The constraint section of this clause limits the first slot position to the values ‘data’ or ‘service’.

Table 12 A structure clause

```
<tpl.structure length="2">
  <tpl.constraint position="0">
    <tpl.value>data</tpl.value>
    <tpl.value>service</tpl.value>
  </tpl.constraint>
  <tpl.specification>
    <tpl.select node="UML:Class">
    <tpl.match nodeType="ATTRIBUTE_NODE" attribute="name" position="1">
    <UML:ModelElement.stereotype>
    <UML:Stereotype>
    <tpl.match nodeType="ATTRIBUTE_NODE" attribute="name" position="0" publish="TRUE"/>

</UML:Stereotype></UML:ModelElement.stereotype></tpl.match></tpl.select>
</tpl.specification></tpl.structure>
```

Table 13 Mutable document nodes

```
<!—Slots:      (tagname      1st_value className   2nd_value)   (xmi.id)        -->
<!—Client use: (PermitAccess fromClass  inClass   toOperation)                   -->

<tpl.roundtrip length="4" auxLength="1" tpl.message=”Processing Access Controls”>
  <tpl.declare position=”0” name=”PERMIT_ACCESS”/>
  <tpl.declare position=”1” name=”VAL_1”/>
  <tpl.declare position=”2” name=”IN_CLASS”/>
  <tpl.declare position=”3” name=”VAL_2”/>
  <tpl.declare position=”4” name=”XMI.ID”/>
  <tpl.constraint position="PERMIT_ACCESS">
    <tpl.value>PermitAccess</tpl.value>
  </tpl.constraint>
  …
<tpl.specification>
  ...
  <UML:TaggedValue tpl.mutable="TRUE">
  <tpl.match nodeType="ATTRIBUTE_NODE" attribute="tag" position="PERMIT_ACCESS">
  <tpl.match nodeType="MULTIPLE_ATTRIBUTE" attribute="value" tagIndex="0"
        length="2" position=”VAL_1”>
  <tpl.match nodeType="MULTIPLE_ATTRIBUTE" attribute="value" tagIndex="1"
        length="2" position=”VAL_2” >
  <tpl.match nodeType="ATTRIBUTE_NODE" attribute="modelElement" position=”XMI.ID” >
    <tpl.selectNode node="UML:Class">
    <tpl.match nodeType="ATTRIBUTE_NODE" attribute="xmi.id" position=”XMI.ID”>
    <tpl.match nodeType="ATTRIBUTE_NODE" attribute="name" position=”IN_CLASS”
        publish="TRUE">
      …
```

The specification searches all the nodes in the XML input for *UML:Class* nodes. For each node of this type it extracts the name attribute, which is unified with the second unification slot. The template then searches child nodes for the stereotype (*UML:ModelElement.stereotype /UML:Stereotype*) and unifies the attribute name of the stereotype with first unification slot. Of course, this slot is constrained, so the only values that succeed are ‘data’ or ‘service’. The effect of this clause, therefore, is to search the XML input for *UML:Class* nodes with a stereotype of ‘data’ or ‘service’ and, depending on mode, publish predicates of the form *(data|service,name)*. The form of this template is very similar to other template languages, demonstrating that although reversible templates are theoretically quite different to conventional templates, their programming form can be made familiar.

Although the structure of the XRound language is a full XML tree, the normal page layout of the template is procedural, except in sections where the tree structure needs to be exposed. This aids comprehension for those familiar with other template languages.

5.2 Mutable nodes

The specification of mutable nodes is essentially the same; Table 13 shows part of a roundtrip template clause for the same application.

The comments at the start of this extract describe the use of the unification slots, and the resulting application predicate; these names are then declared as aliases for the slot positions and later used to define positions. This template matches an XMI tag, which references a UML class. The name of the tag is *PermitAccess* and the tag value has two separated components (e.g. *PermitAccess = ”subject,object”*). The application predicate contains the same information as the tag, but also includes the name of the class in which the tag was declared (*inClass*). The first four template slots correspond to the values in the application predicate, and the fifth is used for the *xmi.id* which references the UML class. The header to this clause specifies the number of unification slots, and constrains the first to the single value ‘PermitAccess’. Note that since this constrained value is part of the predicate exchanged with the application, the use of constants, as described in section 4.3.1, is not appropriate.

The specification navigates directly from the document root (XMI) to a tagged value, which is marked as mutable. This specifies that any tagged values that match this clause will be re-written on export. This navigation identifies all possible tagged values, but only those that unify as far as the ‘publish’ attribute at the end of this fragment will be rewritten.

The next three match statements unify the three elements of the tag (name plus two values) with their respective slots. An important feature of this language is that these statements are able to extract data from the XML source and publish them to the client application, and also obtain predicates from the client and update the XML source, depending upon the operational mode of the template processor.

The fourth match operation unifies the *modelElement* attribute value with an auxiliary slot in the unification template (i.e. one that is not part of the application client’s predicate). This value is the *xmi.id* of the class to which the tag is related; *selectNode* then navigates to the corresponding class by selecting all the class nodes in the XML input, and selecting the one with the correct xmi.id. This involves searching the entire input tree in order to

dereference a single *xmi.id*; a more economical approach is described in section 4.4.

The final match statement unifies the class name associated with this *xmi.id* with the third template slot. At this point the publish attribute tests if the unification process is complete, causing publication to the client, or the addition of a node to the XMI document, depending upon the direction of processing.

In order to write a template, such as the fragment in table 13, it is necessary to understand the relevant parts of the source metamodel, and the predicates required by the application. However, the programmer's view of the process is one of selecting model elements, and specifying how predicates are assembled; these are essentially the only operations that are exposed. This is therefore very similar to standard template processing, where the template specifies location and format. The only features in this fragment that indicate that it is reversible are 'mutable' attributes, which show which nodes can be modified. The programmer must understand that nodes marked as mutable must be fully defined in the application predicates (see section 6.2.2), but in most other respects the semantics of bidirectional processing are hidden from the template programmer, who is still able to think of the reversible template as little more than a 'select and publish' script. (Limitations to processing transparency are illustrated in sections 5.3 and discussed in section 8.3.)

One notable feature of this fragment is the relative lack of constraint checking. In this application the two component values in the tag are known types, the first corresponding to a class of a known stereotype, with a navigable association to the class in which the tag appears, and the second to an operation within that class. It would be quite straightforward to use the template to check that these values correspond to correct types. However, there are good reasons for avoiding these checks at this stage. Firstly, the template is specific to the tool that generated the XML input, but given that the template processor delivers tool-independent predicates, the type checking could be coded once, in the application, rather than separately for each supported tool. The application is able to give meaningful messages about type problems in the application domain, because the predicates adequately describe the model from an application perspective. For example, if a predicate specified a security access permission to a non-existent user role, this problem could be meaningfully described to the user.

This argument also mitigates against the use of the UML Object Constraint Language (OCL) for application-specific type checking, since constraints written within the UML will be specific to a proprietary UML metamodel and file structure. Of course, there is likely to be valuable type-checking within the UML model for constraints that are not related to the analysis domain, and which may not be fully exposed to the application.

There is also a second consideration, which is that in its normal operation the template processor will often fail to unify, since it will attempt to match nodes and predicates that are not compatible. If constraint checking is included in the template, then badly constructed types will not unify, and will not be passed to the application. However, the result of a constraint failure in a template processor is silence, whereas constraint failures in the application can generate meaningful type warnings to the user. The programming philosophy is therefore to specify the minimum in the template language, consistent with establishing an accurate relationship between the XML input and application predicates, and to carry out more application-specific type checking in the application.

5.3 A contrasting XML metamodel

The previous sections provided examples of template clauses from a real application; since this application supports multiple UML tools, it is possible to contrast Table 13 with the template for a different UML source. This provides insight into the extent of the differences between proprietary metamodels, and the role of the template in hiding the application from such differences.

Table 14 delivers the same predicate to the application as Table 13, but for XML documents produced from a different proprietary design tool. Both tools support UML 2 and use XMI as a common exchange format; they are not identified here since they are both are well regarded propriety tools, and identifying them would invite comparisons based only on one aspect of their metamodels.

The template in Table 13 extracts a complete tag from one area of the model, and then looks up the referenced *xml.id* to obtain the associated class name. The structure of the XMI corresponding to the template in Table 14 is quite different; it is the tag definition, not the class definition that is identified with the *xmi.id*. Here, the document is first searched for a tag specification, in order to extract the *xmi.id* for the required tag name ("PermitAccess"). The instance of a tag is in a sub-tree within the *UML:Class* to which it is attached; the tag information is extracted, and the template finally checks that the tag identifier (*xmi.id)* in the class matches the required definition.

This example illustrates the power of the reversible template. From the user perspective both source tools carry out the same function, and both export XMI models. The metamodels are radically different, but these differences are only visible in the template definition; from the application perspective the difference is hidden, and the analysis tool is able to use either source.

Table 14 Mutable document node for a contrasting XMI source

```
<tpl.specification>
  ...
  <UML:TagDefinition>
    <tpl.match nodeType="ATTRIBUTE_NODE" attribute="name" position="PERMIT_ACCESS">
    <tpl.match nodeType="ATTRIBUTE_NODE" attribute="xmi.id" position="XMI.ID">
  <tpl.selectRegisteredNode node="UML:Class">
    <tpl.match nodeType="ATTRIBUTE_NODE" attribute="name" position="IN_CLASS">
    <UML:ModelElement.taggedValue tpl.mutable="TRUE">
      <UML:TaggedValue>
        <UML:TaggedValue.dataValue>
          <tpl.match nodeType="MULTIPLE_TEXT" tagIndex="0" length="2" position="VAL_1">
          <tpl.match nodeType="MULTIPLE TEXT" tagIndex="1" length="2" position="VAL 2" >
        <tpl.moveUp steps="1">
          <UML:TaggedValue.type>
            <UML:TagDefinition>
              <tpl.match nodeType="ATTRIBUTE_NODE" attribute="xmi.idref"
                position="XMI.ID"  publish="TRUE">
      …
```

To support this metamodel it is necessary to regenerate tag definitions, as well as the instances of tags. The fragment in Table 14 regenerates instances of tags, provided the definitions are present. Roundtrip clauses in the template language are executed in the order in which they appear, so in the complete template there is an additional clause to extract or regenerate tag definitions.

The template language therefore has the ability to regenerate models where mutable nodes are mutually dependent, as opposed to simply add or subtract leaf nodes. However, in these more complex examples, bidirectional processing is not fully transparent to the template programmer, who must take into account the order in which the XML model will be regenerated.

6 A Template Processor

The previous sections describe the XRound template language; this section describes a practical template processor which is able to interpret the language, and provide the operations of validation, import and export (round-tripping) of models. The programmatic interface to the template processor is described, showing how template processing is integrated with an application, followed by a brief account of its architecture. This section aims to demonstrate the feasibility of practical processors to support the XRound language, rather than give a full account of implementation issues.

6.1 Template Processor Overview

The XMLSource template processor is a Java class that encapsulates an XML file and allows its client application to import and export predicates from and to an XML document. The design of the processor is given in Fig. 4; although its initial application was to roundtrip XMI documents, there is nothing XMI-specific in XRound, or in this processor.

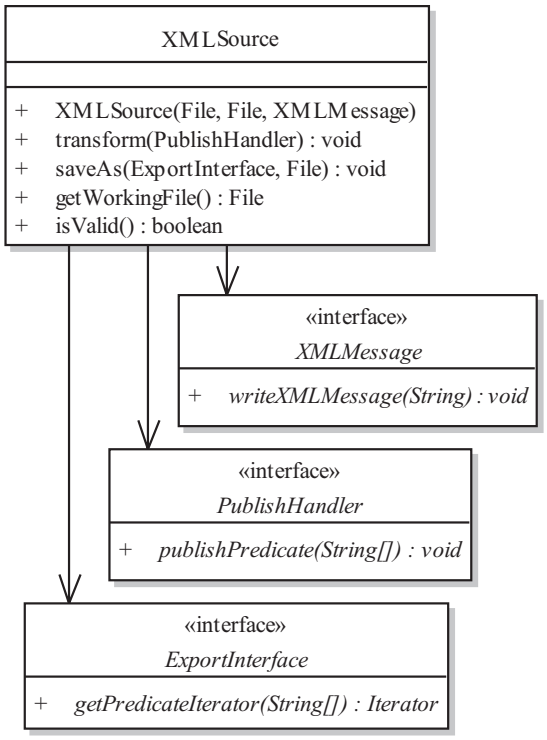

Fig. 4 XMLSource: an XRound Template Processor

The XRound processor is a single class, XMLSource, which encapsulates an XML file whose name is provided to the constructor. Three interfaces are defined in the package, and these call-backs are provided by the application client to allow the processor to import and export predicates.

Predicates are represented as arrays of Strings, such as {class,foo}, which describe features in the XML input that are required by the application. The processor supports three transform operations: *validation*, *import* and *export*. (See Section 3 for further detail.)

Validation. Validation allows the user to check that the template and the input model are compatible. The *XMLSource* constructor takes three parameters, the reference XML File, the Template File, and a message interface. (The Java *File* class encapsulates a file name.) The message interface is used to pass certain error

messages back to the application, particularly those that report inconsistencies between the template and the XML input. A message interface is used in preference to a thrown exception, since it allows a sequence of messages to be reported during processing, which is valuable during template debugging.

The initialization process parses both the Template and the XML input file, and executes the section of the template which is used to validate the input. Methods are provided to allow the client application to check that the validation was successful (*isValid*) and to retrieve the name of the XML input file (*getWorkingFile*).

Import. A single method, *transform*, runs the import process, which extracts predicates from the XML input, as specified by the template, and publishes them to the client application. As each predicate is constructed the *PublishHandler* interface provided by the application client is called to transfer the predicate to the client.

Export. A single output method (*saveAs*) is provided to export predicates from the client application to a named XML file. The output filename is provided by the client, together with an interface (*ExportInterface*) which allows XMLSource to obtain predicates from the application. This callback is slightly more functional than the other interfaces, but is still straightforward: the client is provided with an incomplete predicate, which is an array of Strings, some elements of which may be null. The client responds with an iterator, which encapsulates predicates matching this template.

The *saveAs* method updates the reference XML input with predicates obtained from the application, and then writes the result to the named file. File naming strategies and backup files, etc, are determined by the client application.

Because the input XML is encapsulated by the template processor, there is no need for the complete XML tree to be exported to the application; the transformation therefore includes only the features required by the application.

An important feature of the template processor is its straightforward client interface; this is a direct result of the reversible template model, since:

- The application needs to obtain only the predicates that it requires for its function, the rest of the input XML remains hidden.
- The application interface is independent of the tool used to generate the XML: any tool differences are accounted for in the template.
- The template includes an explicit validation section that is run at initialisation.

A discussion of how the three operations relate to the template specification is presented in section 3.3; the remainder of this section describes implementation issues.

6.2 Template Processor Implementation

The template processor is based around a core recursive structure which alternates between navigating the template, then implementing template instructions, often by traversing the document tree; this core architecture is shown in Fig 5.

The document which is traversed by the template processor is always the XML input model, and this model is retained until it is updated by round-trip processing. In this way any elements of the XML input model that are not transferred to and from the application as predicates are retained.

Each of the main public methods (the constructor, *transform()* and *saveAs()*) sets a processing mode and then calls *processDocument()* once for each clause to be processed. The processing modes used for each method and clause type are listed in table 15.

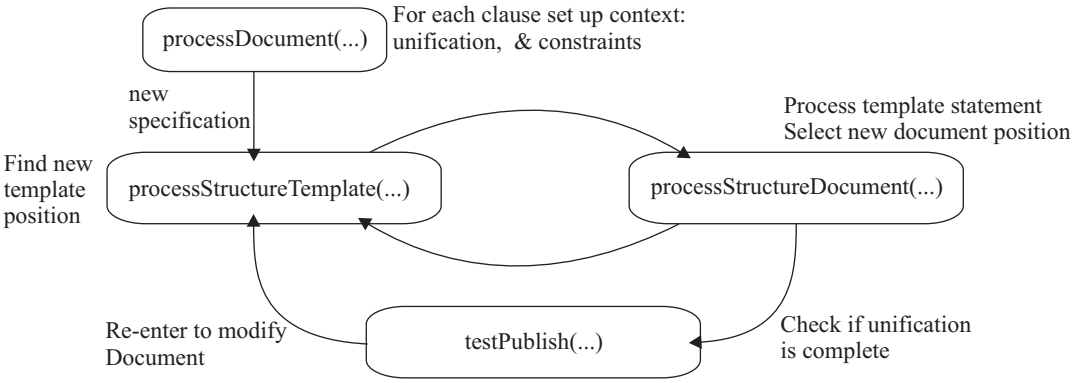

Fig. 5 Template processor core architecture

Table 15 Processing modes and associated clause types

Method	Clause	Mode
XMLSource constructor	validate	validation
transform()	structure	forward
transform()	roundtrip	forward
saveAs()	roundtrip	clean
saveAs()	roundtrip	reverse
testPublish()	see text	reverse2

The processing modes primarily condition the behaviour of the *testPublish()* method, which is used to check if a unification has succeeded, together with some modifications to the behaviour of the core recursive cycle, most notably success and failure handling.

The processing of validation and forward modes is straightforward; the *processDocument()* method reads the header, constraints, and declarations from a template clause, builds the recursive context and then calls *processStructureTemplate()* for each specification within the clause. The processing is carried out by a recursive loop between *processStructureTemplate()*, which selects the next template node, and *processStructureDocument()*, which carries out the template instruction, often by selecting the next position in the XML input document.

The calling parameters of these methods are identical, consisting of a recursive context which represents the current position in the combined template and document trees; the main elements of this context are:

- the processing mode;
- the template tree node;
- the document tree node;
- the *Predicate* object (see unification, below);
- the current debug message; and
- a node list (see roundtrip processing, below).

processStructureDocument() is essentially a case statement for each of the possible template operations, each of which is implemented in a separate method; the only method of architectural interest is *testPublish()*, which is invoked whenever a *publish* attribute is encountered in the template. Its behaviour depends on mode. In validation and forward modes it checks for a completed unification, and if this succeeds it either returns success, or publishes a predicate to the application using a callback interface, respectively. Interesting aspects of the implementation are the management of unification, and the use of the core processing architecture for round-trip processing.

6.2.1 Unification

Unification is straightforwardly encapsulated in an inner class (*Predicate*), with the number of unification slots set by its constructor. Class methods include:

- *addConstraint*: specify a constraint on a slot;
- *addvalue*: unify a given value with a slot;
- *getPredicate*: unify and return predicate;
- *indexMatch*: check index between pair of slots.

In general, unification is performed as each new value is added, returning success or failure; however, the use of the *getPredicate()* accessor allows the class to defer the unification of constraints, including the generation of unique names, if required.

6.2.2 Round-trip processing

Round-trip processing updates the XML document to be compatible with the model held by the application; model updating is only carried out in areas of the model that are identified by nodes in *roundtrip* template clauses which are marked as mutable. Two passes of the complete template are made, the first cleans the retained XML input document by removing mutable nodes, and the second (reverse) regenerates any such nodes that are present in the application model. All other elements of the XMI input document are preserved. This imposes a constraint on the template programmer, that the contents of mutable nodes must be preserved in the application predicates. Constructing the processing in this way allows the application to signal that nodes should be completely deleted from the model, simply by deleting the relevant predicate(s).

In both modes, the *processStructureDocument()* method records the context of mutable document nodes; this node list is built as part of the recursive context, and eventually passed to the *testPublish()* method. Essentially, nodes are recorded as potentially mutable, but no action is taken until unification is confirmed.

In the clean mode, *testPublish()* uses the node list to build a list of document nodes to be deleted; they are actually deleted after the whole template has been processed.

A similar process is used to regenerate mutable modes; however, the generation of new nodes is slightly more complex. If *testPublish()* determines that a unification is successful, then the recursive context contains a node list which must be created. This is achieved by re-entering the main recursive loop, once for each new node, using the mode *Reverse2*; this provides *processStructureDocument()* with sufficient information to rebuild any nodes that are missing. This re-entrant recursive structure has proved to be an elegant and economical solution for reverse processing.

7. A Practical Application in Security Analysis

The XRound language and template processor have been successfully used to support security analysis of UML models. This section briefly describes this practical experience.

7.1 The application background

The Security Analyst Workbench (SAW) supports risk-based security analysis and design; *analysis*, is concerned with determining the risks in a system, and *security design*, is the specification of control requirements that mitigate those risks. SAW is part of a suite of models, analysis methods and tools known as the Security Design Analysis framework (SeDAn) [27]; the framework will not be described here, however, its use in industrial applications [26] necessitated competent tool support.

Security risks are potential threat paths through a system, from attackers to assets; to carry out risk analysis it is necessary to assemble a single system model, which includes:

- the functional design of the system;
- security requirements (such as access controls) that specify how the system is protected;
- system users, including administrative organizations;
- security goals, and unwanted outcomes for specific assets (e.g. loss of integrity of a particular data item), and their impact in business terms; and
- attackers, their goals, and the likelihood of attack.

One objective of the Security Analyst Workbench is to integrate security design with standard system engineering practice; for this reason these elements of the system model are divided into three main categories:

- the functional design, which is a standard engineering design in UML, usually in the form of a Platform Independent Model;
- security requirements, which are attached to components in the system design and become specialised properties of the design model; and
- the security environment (attackers, security goals, asset concerns etc) which is specified in an auxiliary model.

The security analyst uses a standard functional system design, builds a complementary specification of the security environment, and then analyses the resulting model for risks. Security requirements are specified to manage the risks, and these become part of the functional specification to be implemented.

The specification of security requirements is a *design* activity: the process of establishing a protection strategy involves choices about the placement and type of security requirements; risk analysis informs that choice, it does not automate it. As a result, SAW provides a richer and more interactive set of user functions than are suggested by 'analysis', and many of these functions are concerned with interactively managing security requirements.

There are therefore two equally important functions for the security analysis tool: risk analysis and requirements management. The latter involves creating and testing new model properties within the analysis tool, resulting in the need to update the UML documentation. Security analysis in practice [26] and requirements specification [28] are described in more detail elsewhere.

7.2 Model Management Requirements

The Security Analyst Workbench (SAW) is a specialised analysis tool, which uses a system design expressed in UML and created using a separate design tool. Security requirements are set and modified during analysis, and these must eventually be reflected in the reference system design.

The issues exposed in this way will be common to many specialized analytic tools; they include:

- a separate specialized and ephemeral model on which the actual analysis is conducted;
- the need for model management facilities within the analysis tool; and
- the need to propagate property changes back into the primary UML documentation.

The solution adopted for SAW is to use XMI as its persistent model format, and furthermore, to use whatever UML metamodel is native to the user's development environment. The import and export mechanism between XMI and the analysis model must be readily adapted to different UML sources, and must ensure consistency in the round-trip operation. These requirements are well supported by, respectively, a template processor, and a single template specification that can be used for processing in both directions, thus ensuring round-trip consistency.

The XMLSource template processor has successfully supported SAW through several iterations, in which new functionality has been added, including new model properties and new templates for different proprietary UML design tools. Our practical experience is that even well regarded tools conforming to XMI may have considerable differences in their metamodels, confirming the need for an import/roundtrip mechanism that decouples the application from the source metamodel. A practical example of this problem is presented in section 5.

Given the performance management features in XRound, the performance of the template processor is dominated by the time taken to open and read XML documents, rather than template processing. From the overall system perspective the user experience is considerably enhanced by using a reversible template: model properties can be changed and tested within the analysis tool. If, instead, a unidirectional import process was used, it would be necessary to change properties in the UML design tool and re-transform the model for analysis to evaluate the effect of each change; this would represent a considerable processing overhead to change and test model properties, which is inconsistent with the need for iterative analysis.

The design philosophy for XRound was to create a minimal set of features consistent with a practical language and add more complex (e.g. higher order) features if they could be shown to be necessary. The design iterations in SAW have tested the language specification by adding new model properties, requiring different parts of the source metamodels to be regenerated, and by accommodating

different proprietary source metamodels. This has been achieved without significant addition to the core language, although it has been necessary to remove some implementation restrictions in the template processor[3]. The programming philosophy of minimising the type checking carried out by the template (see section 5.2 for discussion and rationale) is perhaps one reason why a relatively simple template language has proved sufficient.

In summary, practical experience has demonstrated the need for a programmable approach to roundtrip transformation of models, and the effectiveness of the XRound template language.

7.3 Worked Example

This section provides a simple worked example in which a Platform Independent Model is enhanced with a description of its security environment, a security policy established and tested, and the UML system model updated with the resulting security requirements. The purpose of the example is to illustrate how reversible processing is used to support a security workflow. For readers interested in more technical details, accounts of requirements modelling [28] and analysis [29] are published elsewhere, and a full account of the metamodels, profiles and security requirements is also available [27].

The system fragment used in this example is shown in Fig. 6; it specifies an office system, with business data (*OfficeData*) which is managed by a service (*OfficeServer*). In this example the service provides only one operation (*update*), to allow a manager to modify the data. The service is normally accessed by an internal client (*Home*) to which business managers have access. The complete system also has connectivity to the internet, which may include services (*FreeSoftware*) that provide access to software of unknown provenance (*Games*) and publicly available clients (*PublicAccess*). Of course, the actual system would have many more operations, this limited functionality is chosen to limit the number of security requirements needed for the sake of example.

The stereotypes in this system denote entities that provide a business service (<<*service*>>), services that may be directly accessed by users (<<*client*>>), and an association stereotype (<<*managed*>>[4]) that indicates that data is bound to a particular service.

This is a Platform Independent Model, since no commitment to an implementation has been given: the services could be implemented by people, business departments or, more usually, some form of information system. Specifically, no binding between services and platforms or platform types is assumed.

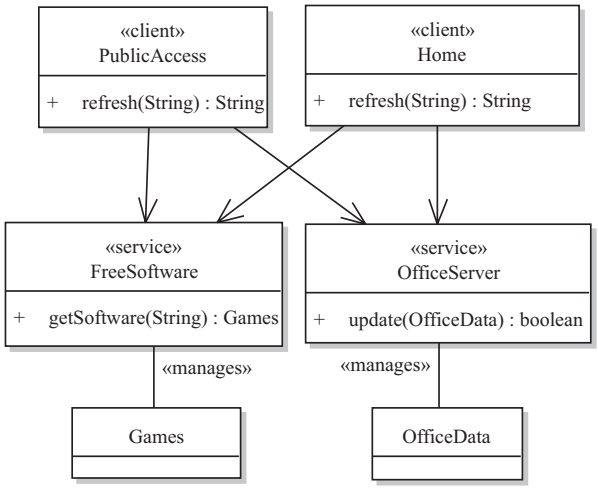

Fig. 6 Example System

Security risk analysis involves finding paths from attackers to assets of concern, where the concern is to avoid a particular unwanted outcome. Before the security characteristics of features of this system can be analysed it is necessary to specify the security environment in which the system operates, including the identification of potential attackers and associated assets. The security environment effectively forms the baseline assumptions for the security analysis, so it must also be recorded with the system documentation in the UML model; an example of such an environmental specification is given in Fig. 7.

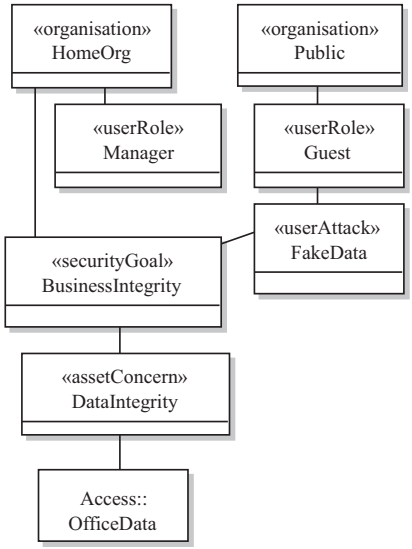

Fig. 7 Security Environment

Fig. 7 specifies the organisations (the target business *HomeOrg*, and the *Public*), and their associated user roles (*Manager*, *Guest*). This system has a single security goal (*BusinessIntegrity*) which is linked via an association class (*DataIntegrity*)[5] to the only asset of concern (*OfficeData*), which appears in the system model. Much of this information is usually available in system requirements

[3] For example, the first release of XMISource did not support *tpl.moveup* within mutable nodes.

[4] This is used to distinguish between different objectives for data protection in security modelling, but does not play a part in this example, because the characterisation of different threat paths is not described here, see references for more information.

[5] The purpose of the association class is not evident in this example; it carries attributes that are specific to assets or groups of assets, such as the impact of the attack.

documents or use cases, and generally does not need to be created specifically for security purposes; however, the attackers also need to be identified together with their objectives. In this example the *Guest* user may attack the *BusinessIntegrity* goal, meaning that any assets linked to that goal may become the target of attack. Attacks may originate with user roles, organisations or outsiders, all of which can be modelled similarly.

It is evident from the security environment and the system design that some security requirements will be needed, in particular access controls that prevent *Guest* users of the *PublicAccess* client using the *update()* operation to modify *OfficeData.* Since the system model has not been bound to particular platforms it is also necessary to record the assumption that the *Home* client is situated within the business, specifically that *Guest* users do not have access to this management client. This will become a constraint on how the system is bound to platforms, and is described as *Deployment Constraint*. In this example the constraint must specify that the *Home* client is never bound to a *Public* platform – i.e. one administered outside the business.

This informal analysis suggests the need for the security policy given in table 16.

Table 16 Initial Security Policy

```
Home
   Deployment Constraint: Public

OfficeServer
   Access Constraint: from PublicAccess
   client to any operation
```

The security analyst has a choice at this point, the policy can be recorded manually in the system design, by adding tags that will in due course constrain the implementation and operation of the system, or they can be added using the security analysis application. The latter is provided with an interactive capability for setting and changing policies, so this is often more convenient, but either is possible.

The next step is to check that this policy is sufficient, and for that purpose the XMI representation of the UML model is imported into the analysis tool, using a template as described in this paper. The XMI[6] representation of this example contained 1479 lines of XML, but the security related elements were encoded in just 50 predicates. The XMI contains a wealth of information about the specific UML tool, its working properties, and diagram layouts that are not needed for security analysis.

The security analysis tool is able to carry out a range of different analysis functions, but the most basic is to determine if there are any threat paths in the system. This analysis discovered an unanticipated threat, shown in table 17.

Table 17 Analysis Result

```
DataIntegrity path trace:

Operation PublicAccess.refresh(in)
called from: Guest

Operation FreeSoftware.getSoftware(in)
called from: PublicAccess

OperationFreeSoftware.getSoftware(return)
called from: Home

Operation OfficeServer.update(in)
called from: Home

Managed Data OfficeServer/OfficeData
```

Essentially, *Guest* users are able to modify *Games* software, which is then imported by the *Home* client, and is able to subvert the integrity of the business system: a relatively common attack scenario.

Normally, access policies are located at the service and manage the access from remote clients; however, this threat requires the *Home* client to ensure that it does not access potentially dangerous external data: the constraint must be enforced by the client, not the service. This type of policy is distinguished as a *RefuseToAccess* requirement, because its implementation is likely to be different from normal access control policies.

The analyst adds the *RefuseToAccess* policy to the model using the interactive policy editor which is part of the security analysis tool, and then re-runs the threat analysis to check that there are no remaining threat paths in the system. Practical experience has resulted in both the policy management and analysis functions being combined within a single analysis tool, because in complex systems the analyst may need to try a range of different security strategies before deciding on a coherent policy. In many proprietary UML tools, exporting a UML model as XMI is relatively slow, so testing security policy variations by changing and exporting the UML model is not consistent with an interactive security design workflow.

When the analyst has decided that the security policy is appropriate, the requirements established in the analysis tool are re-integrated into the UML model, using the same reversible template that was used to extract the data for analysis.

Security requirements established in the security analysis tool are added as tags to relevant classes within the UML model, in order to constrain their implementation. The tags resulting from the security requirements described above are are given in table 18. (Access permissions are also added to the other services for completeness, but this is outside the scope of this discussion.)

[6] This XMI was generated using the Enterprise Architect UML tool.

Table 18 Security Tags Added to System Model

```
Home Client
NoDeploy = Public
RefusetoAccess=FreeSoftware.getSoftware
OfficeServer
PermitAccess=Home, ALL_OPERATIONS
...
```

The forgoing example has described the security analysis and design cycle. In this case three different types of security requirement have been recorded in the PIM, and these are used in different ways in the subsequent implementation lifecycle. Deployment constraints limit how the PIM can be bound to concrete platforms; access permissions will be carried forward in the implementation and be issued as policies to be interpreted by access management infrastructure; and Access Refusals are similarly carried forward to be used as infrastructure policies, but are distinguished since they will be implemented by a different architectural binding.

This example has illustrated how security analysis and design is integrated into Model Driven Development, and how this is facilitated by the reversible template language. The analysis and design process described here has been implemented and used in practice; the automated use of these security requirements within the development lifecycle is planned as future work.

8. Limitations

Practical limits arise from variability in XML source metamodels, the scope of the template language, and the implementation of the template processor.

8.1 Source metamodels

Differences in metamodels between UML tools is a well known problem, was one of the main motivating factors in the design of XRound, and has been mentioned at several points in the paper. Different templates are required for different UML tools, but the use of a reversible template isolates the application logic from this variability. The XML import behaviour of tools can also vary in detail; for example, some tools regenerate missing *xmi.id* fields, where others fail. The design of a template may therefore go beyond the need to understand (part of) the source metamodel. Although this is an inconvenience, it has not yet proved a major problem, or required tool-specific language features.

8.2 Language features

There are two aspects of the language that could be considered as candidates for enhancement:

- the performance management mechanism; and
- the unification scheme.

The performance management mechanism is essentially a cache, so it is natural to ask if the cache could be built transparently, without user involvement. Such a mechanism would be feasible; however, allowing explicit performance management in the language allows finer control by the user than would be possible in an automated system. For example, an automated cache would need to be conservative in the sense that it would need to cache all possible nodes of a given type, whereas it may be possible for a programmer to be more selective. In summary, automated caching is a possible enhancement to the template processor, but even with such a feature it is desirable to retain the performance management elements in the language.

The unification scheme could be enhanced to allow more sophisticated forms of logic; for example:

- allowing more general logical constraints on unification; and
- regular expressions for matching or extracting of elements of an attribute or text value.

More general unification constraints were not designed into the language at the outset because of the programming philosophy, discussed in section 5.2. In brief, the programming objective is to avoid complex type checking in the template, since it is better implemented in the application.

In XML the fields (attributes and text fields) should already be atomic; however, it was evident from the first applications that the atomicity of XML fields cannot be relied on, so some mechanism is required to identify components within fields. Any component matching mechanism, however, must also be able to allow the fields to be incrementally recreated, when the XML is regenerated, and this is not a property of an arbitrary regular expression. The simple component parsing approach in the language was designed to allow roundtrip reconstruction of such fields. The design of regular expression languages that allow incremental pattern building as well as extraction is an open question.

In brief, it is possible to envisage new language features that offer more logical complexity; however, any such features need to be reversible, and their design is not therefore straightforward. Practical experience has not yet indicated the need for such features.

8.3 Template processor

The problem of interdependent mutable nodes was mentioned in section 5.3; in this example a tag definition within a UML class could not be created unless there was a tag declaration elsewhere in the XML document to bind a tag name to an *xmi.id*. The language is able to support constructs of this type, so the models created in the roundtrip can be modified in structure, not merely by the addition or removal of leaf nodes. However, this process is not transparent to the programmer, who must order the roundtrip clauses to ensure that the document can be built incrementally.

It would be desirable for the template processor to implement a more transparent approach to rebuilding more complex models; unfortunately there are open technical problems in achieving this. For example, it is not clear that the detection and resolution of cyclic dependencies between mutable nodes is feasible within reasonable complexity bounds. Future development in this problem area is likely to inform the template processor, rather than the language itself.

9. Conclusion

XRound adds a new dimension to the template processing of XML models: the ability to transform data in both directions with a single descriptive template. Reversible template processing solves the problem of maintaining independence between XML source documents and analytic tools, while retaining the benefit of easily scripted transformations. Reversible templates could provide a clean implementation mechanism for bidirectional transformations specified in QVT, and could help in the definition and implementation of model merging languages as well.

This paper outlines the requirements of specialised analytic tools, the theory behind reversible templates, and presents a mature template language, XRound. This language is supported by a template processor, and includes performance management and debugging facilities.

The examples presented here illustrate the extent that the underlying semantics of unification and reversible transformation are hidden from the template programmer, who is usually able to think of the template as a 'select and publish' script.

The language and its processor have been used in practice to support security analysis. The application is a specialised analytic tool which supports the rigorous security risk analysis of UML models, usually PIMs, and provides an environment in which the user can interactively set and test security requirements. Properties established during analysis are re-integrated into the engineering documentation (i.e., the UML models) using a bidirectional transformation specified by the XRound template language. The use of XRound has isolated the need to support different source metamodels from the analysis application.

Practical experience to date has not indicated any major weaknesses in the language or the processor; however, section 8 discusses a number of possible enhancements to the language or processor, such as the use of regular expressions to match XML fields, or improving the transparency of bidirectional processing. Because of the requirements of bidirectional processing these options are research questions, rather than simple language enhancements, highlighting the need for further research into reversible programming constructs.

References

[1] Model Driven Architecture (MDA), Object Management Group, Specification ormsc/01-07-01, 2005.

[2] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Object Management Group, Specification ptc/07-07-07, 2007.

[3] ATL : Atlas Transformation Language, ATLAS group (LINA & INRIA), available at http://www.eclipse.org/gmt/atl/doc/ (accessed January 2007), 2005.

[4] XMF Reference Guide, Xactium Ltd, UK, 2005.

[5] O. Patrascoiu, YATL:Yet Another Transformation Language, Proceedings of the 1st European MDA Workshop (MDA-IA), available at http://www.cs.kent.ac.uk/pubs/2004/1829 (accessed January 2008), 2004, pp. 83-90.

[6] A. Balogh, D. Varro, Advanced Model Transformation Language Constructs in the VIATRA2 Framework, Proceedings of the Symposium on Applied Computing (SAC'06) - Model Transformation Track, ACM Press, 2006, pp. 1280-1287.

[7] D. S. Kolovos, R. F. Paige, F. A. C. Polack, The Epsilon Object Language (EOL), Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications, Springer, Lecture Notes in Computer Science Vol 4066, 2006.

[8] J. R. Cordy, I. H. Carmichael, R. Halliday, The TXL Programming Language, Version 10.5, Software Technology Laboratory, Queen's University at Kingston, Ontario, available at http://www.txl.ca/ndocs.html (accessed January 2008), 2007.

[9] MOF Model to Text Transformation Language RFC, Object Management Group, Specification ad/04-04-07, 2007.

[10] J. Manning, Code Generation in Action, Manning Publications, 2003.

[11] Velocity User Guide, Ja-Jakarta Project, available at http://www.jajakarta.org/velocity/velocity-1.4/docs/vtl-reference-guide.html (accessed January 2008), 2007.

[12] Eclipse's Java Emitter Templates (JET), IBM, 2004.

[13] L. Tratt, The Converge Programming Language, Department of Computer Science, King's College London, Technical report TR-05-01, 2005.

[14] M. Alanen, I. Porres, The Coral Modelling Framework, Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004, Turku Centre for Computer Science, General Publication Vol 35, 2004.

[15] Z. Hu, S.-C. Mu, M. Takeichi, A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations, Proceedings of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, ACM Press, 2004.

[16] R. A. Pottinger, Merging Models Based on Given Correspondences, Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann, 2003, pp. 826-837.

[17] M. Didonet Del Fabro, B. Jean, J. Frédéric, B. Erwan, G. Guillaume, AMW: A Generic Model Weaver, Proceedings of the IDM'05, Premières Journées sur l'Ingénierie Dirigée par les Modèles, 2005.
[18] J. Foster, M. Greenwald, J. Moore, B. Pierce, A. Schmitt, Combinators for Bi-directional Tree Transformations, ACM Transactions on Programming Languages and Systems. 29(3) (2007).
[19] M. Buchholtz, S. Gilmore, V. Haenel, C. Montangero, End-to-end integrated security and performance analysis on the DEGAS Choreographer platform, Proceedings of the Formal Methods 2005, Springer-Verlag, Lecture Notes in Computer Science Vol 3582, 2005.
[20] D. Petriu, et al., Performance Analysis of Security Aspects in UML Models, Proceedings of the Sixth International Workshop on Software Performance (WOSP 2007), ACM Press, New York, USA, 2007, pp. 91-102.
[21] S. H. Houmb, G. Georg, J. Jurjens, R. France, An Integrated Security Verification and Security Design Trade-off Analysis Approach, in: H. Mouratidis and P. Giorgini (Eds.), Integrating Security and Software Engineering, IGI Global, 2006.
[22] R. Breu, M. Hafner, B. Weber, A. Novak, Model Driven Security for Inter-organizational Workflows in e-Government, in: E-Government: Towards Electronic Democracy, Springer Berlin, 2005, pp. 122-133.
[23] J. Jürjens, Secure Systems Development with UML, Springer Berlin, 2005.
[24] G. Brændeland, K. Stølen, Using model-based security analysis in component-oriented system development, Proceedings of the 2nd ACM Workshop on Quality of Protection (QoP '06), ACM Press, New York, USA, 2006, pp. 11-18.
[25] J. Clark, XSL Transformations (XSLT) Version 1.0, W3C, Recommendation 1999.
[26] H. Chivers, M. Fletcher, Applying Security Design Analysis to a Service Based System, Software Practice and Experience: Special Issue on Grid Security. 35(9) (2005) 873-897.
[27] H. Chivers, Security Design Analysis, in Department of Computer Science. 2006, The University of York: York, UK. p. 484.
[28] H. Chivers, J. Jacob, Specifying Information-Flow Controls, Proceedings of the Second International Workshop on Security in Distributed Computing Systems (SDCS) (ICDCSW'05), IEEE Computer Society, 2005, pp. 114-120.
[29] H. Chivers, Information Modeling for Automated Risk Analysis, Proceedings of the 10th IFIP Open Conference on Communications and Multimedia Security (CMS 2006), 2006.